

Correcting Two Deletions and Insertions in Racetrack Memory

Alireza Vahid, Georgios Mappouras, Daniel J. Sorin, Robert Calderbank
Department of Electrical and Computer Engineering
Duke University

Abstract—Racetrack memory is a non-volatile memory engineered to provide both high density and low latency, that is subject to synchronization or shift errors. This paper describes a fast coding solution, in which “delimiter bits” assist in identifying the type of shift error, and easily implementable graph-based codes are used to correct the error, once identified. A code that is able to detect and correct double shift errors is described in detail.

Index Terms—Racetrack memory, synchronization error, shift error, insertion/deletion channel, shift error correcting codes.

I. INTRODUCTION

Historically, improvements in memory and storage have been due to advances in the memory technologies themselves together with innovations by computer architects who design memory and storage systems, and by coding theorists who design codes for storing data.

Racetrack memory is a non-volatile memory engineered to provide both high density and low latency, and aims to replace conventional memories such as DRAM and Flash. Racetrack memory stores data in “tape-like” tracks, and to read a stored bit, an electric current is injected to place the desired bit under the read-write port. Competing technologies, such as phase-change memory (PCM) and magneto-resistive random-access memory (MRAM) cannot match the density of racetrack memory. In terms of latency, only SRAM (which is a volatile technology) has a slight advantage, but it comes at the cost of lower density [1]–[3].

Racetrack uses current injection to *shift* magnetic domains that store data bits on these tracks. In this technology, a deletion occurs when the injected current is larger than expected causing one or more domains to be skipped. A repetition occurs when the injected current is smaller than expected, so that the domain under the port does not change, and we read the same bit two or more times. We refer to deletion and repetition errors as shift errors. These types of synchronization error define a channel that has been studied extensively by information theorists [4]–[10]. There is no closed form expression for the capacity of this insertion/repetition and deletion channel, but standard linear codes are known to be far from optimal. For example, the rate penalty for correcting a single shift error is logarithmic in the block length, but the optimal linear code has rate $1/2$ [11].

There is some prior work in the computer architecture literature that seeks to handle shift errors by introducing extra read/write ports. The HiFi scheme presented in [12] requires

at least two extra reads and one extra write to access a stored bit. When an error is detected, it is corrected by reversing the current injection and reading the memory again. The extra read/write operations and the need for additional hardware limit the attractiveness of this solution and diminish the appeal of racetrack memory.

We propose an alternative solution, based on codes that detect and correct shift errors. The codes are easy to implement, and the circuits for encoding and decoding consume little power. Our coding solution does not require additional hardware, and it does not introduce additional reads or writes to memory.

Section IV describes a product code that can detect and correct up to two shift errors every $m + 3$ shift operations in racetrack memory. The product structure makes it possible to decouple error detection from error correction, so that in the common case of no shift error, the access to stored data is fast. We detect up to two shift errors by combining Varshamov-Tenengolts (VT) codes [13] with blocks of delimiter bits. Our product construction extends the utility of VT codes, which are limited to correcting single shift errors, and we also provide a low complexity construction for these codes. The inner code detects up to two shift errors within a single track, and is able to correct one. The outer code connects data stored on different tracks, and makes it possible to correct two shift errors.

II. RACETRACK MEMORY

In this section we introduce racetrack memory and our error model.

A. Racetrack Background

Racetrack memory stores data in r parallel tape-like tracks. Each data bit is stored in a magnetic domain and neighboring domains are separated by a domain wall as depicted in Fig. 1. All read/write ports and the physical substrate are fixed in position, and as current is passed through the track, the domains pass by magnetic read/write ports positioned near the wire.

Racetrack memory suffers from deletion and repetition errors. Bits are stored on a track and are accessed by injecting a current to shift them and place them under the read/write port. A *shift* is therefore the injection of the current in order to place the next bit under the read/write port. If the injected current is larger than expected, then we might skip one or multiple magnetic domains resulting in *deletion errors*. On the

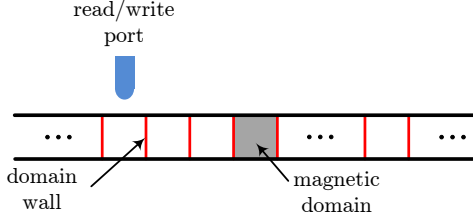


Fig. 1. A single track with one read/write port. One bit is stored per domain and two neighboring domains are separated by a domain wall.

other hand, if the injected current is smaller than expected, the port's position might not change and we might read the same bit twice. We refer to this error as a *repetition error*. We refer to deletion and repetition errors as “shift errors.”

In deletion channels as studied in information theory, the size of the output is equal to the size of the input minus the number of deletions. However, in racetrack memory, the memory controller is always going to provide the desired number of bits, say n , regardless of whether an error occurs, so that an error determines which n bits are provided. This is an important characteristic of racetrack memory, and a consequence is that we first need to determine whether an error has occurred.

There is a difference between a *repetition* and an *insertion*. An insertion means introducing a new bit in a string regardless of the value of its preceding bits. However, a repetition requires the new bit to be equal to the bit right before it. Thus, a repetition error is an instance of an insertion error.

Our goal is to devise a practical code that would allow reliable data recovery in the presence of shift errors in racetrack memory, and for that we need to define the error model.

B. Error Model

We make the following assumptions:

- 1) In each track and in every $m + 3$ shifts (shift is defined above), at most **two** shift errors may occur.
- 2) When reading r parallel tracks, in every $m + 3$ shifts (per track), at most one track may have two shift errors.

As we see later, if the second assumption is violated, then we can detect the errors but we will only be able to correctly decode the data on the tracks that have a single shift error. A more restrictive model for deletion and insertion channels with segmented errors was used in [14] in which only one shift error could happen per segment (a segment is a fixed number of input bits). Moreover in [14], authors assume either deletion errors or insertion errors but not a mixture of the two may occur in the channel; we impose no such constraint.

III. MAIN RESULT

In this section, we present our main contribution. First, we need the following definition.

Definition 1. *If any k input data bits can be mapped to some m -bit codeword such that the k data bits can be recovered*

error-free under the assumptions of Section II-B, we say a zero-error rate of k/m is achievable.

Theorem 1. *For racetrack memory with the error model defined in Section II and with r parallel tracks, we can achieve a zero-error rate of*

$$R_0(\ell, r) = \frac{r-1}{r} \times \frac{2^\ell - \ell - 1}{2^\ell + 6}, \quad \ell \in \mathbb{Z}^+, \quad (1)$$

where $k = 2^\ell - \ell - 1$ is the number of input data bits that are mapped to $m = 2^\ell + 6$ magnetic domains on the memory.

Our code has three main components in two dimension. First dimension is within a single track where we use Varshamov-Tenengolts codes to store data. These codes are able to correct single shift errors. However, Varshamov-Tenengolts codes are not capable of detecting or correcting two shift errors. We enhance these codes by appending some helper bits that we refer to as *delimiter bits* which allow for double-shift-error detection per track. The role of the delimiter bits is to convert racetrack to a traditional deletion/insertion channel. The second dimension is across the r tracks where we deploy a single parity-check code to correct double shift errors. The challenging part is to detect two shift errors and correct one. We use a simple outer code to correct two shift errors.

The rest of the paper is dedicated to the proof of Theorem 1. We also note that $R_0(\ell, r) \rightarrow 1$ as $\ell, r \rightarrow \infty$. As an example, consider $\ell = 6$ (i.e. 57 input data bits) and $r = 8$. Then, we have $R_0(6, 8) = 0.7125$.

IV. PROOF OF THEOREM 1: PRACTICAL CODES FOR RACETRACK MEMORY

In this section, we introduce our code for racetrack memory and we prove Theorem 1. We start with a single track and we devise a code that can correct a single shift error and detect up to two shift errors every $m + 3$ shifts. We then show how to correct two shift errors using an outer code that takes advantage of the spatial diversity in racetrack memory.

On a single track, we encode the data bits into an *extended codeword* of length m which consists of a Varshamov-Tenengolts code (defined below) of length $n = m - 6$ followed by six helper bits that we refer to as delimiter bits.

Definition 2. *The Varshamov-Tenengolts code $VT(n)$ is the set of all binary strings (c_1, c_2, \dots, c_n) satisfying*

$$\sum_{i=1}^n i c_i \bmod{n+1} \equiv 0, \quad (2)$$

where the sum is evaluated as an ordinary rational integer¹.

Remark 1. *VT codes were introduced in [13] to correct errors on a Z-channel. It follows from (2) that these codes are nonlinear over the binary field.*

Encoding: It is possible to construct $VT(n)$ codes for any n , but to maximize the speed of encoding and decoding

¹Technically speaking, our definition is a specific instance of VT codes. In general, VT codes satisfy $\sum_{i=1}^n i c_i \bmod{n+1} \equiv a$, and we used $a = 0$.

Observation						Checksum mod $n + 1$	Decision
position $m - 5$	position $m - 4$	position $m - 3$	position $m - 2$	position $m - 1$	position m		
1	1	0	0	X	X	$= 0$	no error
1	1	0	0	X	X	$\neq 0$	1 del. and 1 rep.: erasure
1	0	0	0	X	X	X	1 deletion error
X	1	1	0	0	X	X	1 repetition error
0	0	0	0	X	X	X	2 deletion errors: erasure
X	X	1	1	0	0	X	2 repetition errors: erasure

TABLE I

RESULTING OBSERVATIONS IN POSITIONS $m - 2$, $m - 1$ AND m FOR ALL POSSIBLE ERROR COMBINATIONS AND X INDICATES AN IRRELEVANT ENTRY.

we choose $n = 2^\ell$ and $k = n - \ell - 1$, because when $n + 1 = 2^\ell + 1$ the necessary modular arithmetic can be executed more efficiently (see [15]). The encoding algorithm is described below.

- 1) Start with a zero-vector \mathbf{c} of length $n = 2^\ell$.
- 2) Set positions that are not powers of two to data bits (there are $k = n - \ell - 1$ such positions).
- 3) Set s to be the minimum value that needs to be added to the checksum $\sum_{i=1}^n i c_i$ to make it equal to 0 modulo $n + 1$.
- 4) Set the $\ell + 1$ positions that are powers of two to the binary expansion (of length $\ell + 1$) of s . Start from c_1 and set it to the least significant bit of the binary expansion of s , and move all the way to c_n and set it to the most significant bit of the binary expansion of s .

It is straightforward to verify that the resulting codeword satisfies (2). We know that in step 3 we have $0 \leq s \leq n$. Since $n = 2^\ell$, the binary expansion of s is at most $\ell + 1$ bits long.

After encoding the k data bits into a codeword \mathbf{c} of size n , we append six delimiter bits to create the *extended codeword* of length $m = n + 6$ that we write to the memory. The six delimiter bits are fixed and equal to **110000**.

In the remainder of this section, we show that the extended codeword introduced above can correct a single shift error and detect up to two shift errors every $m + 3$ shifts. Then, we show how we can correct two shift errors. We present our argument in five steps. In steps 1 and 2, we show that VT codes are capable of correcting a single shift error. Step 3 shows how we can detect up to two shift errors and takes into account the fact that the memory controller is always going to provide the desired number of bits, regardless of whether an error occurs. Step 4 show how to correct a single shift error using our extended codeword. Finally, step 5 incorporates spatial diversity to correct two shift errors.

Step 1: VT codes are single-deletion-error correcting codes. Consider two binary strings $\mathbf{x} = x_1, x_2, \dots, x_n$ and $\mathbf{y} = y_1, y_2, \dots, y_n$. Let $D_{-1}(\mathbf{x})$ and $D_{-1}(\mathbf{y})$ denote the set of all the binary strings of length $n - 1$ that result from a single deletion in \mathbf{x} and \mathbf{y} respectively. If we wish to use these two strings to store different values in a single-deletion channel,

we need

$$D_{-1}(\mathbf{x}) \cap D_{-1}(\mathbf{y}) = \emptyset. \quad (3)$$

The following lemma shows that VT codes are single-deletion-error correcting codes.

Lemma 1 ([16], [17]). *Any two binary strings $\mathbf{x}, \mathbf{y} \in VT(n)$, satisfy (3).*

For completeness, we present the proof of Lemma 1 in Appendix A.

Step 2: VT codes are single-insertion-error correcting codes. Note that a single-insertion-error correcting code is a single-repetition-error correcting code as well. Similar to what we described above, let $D_{+1}(\mathbf{x})$ denote the set of all the binary strings of length $n + 1$ that result from a single insertion in \mathbf{x} . We have the following result that shows the Varshamov-Tenengolts codes can be used for single-insertion-error-correction as well as single-deletion-error-correction.

Lemma 2. *If for two binary strings of length n condition (3) is satisfied, then we have*

$$D_{+1}(\mathbf{x}) \cap D_{+1}(\mathbf{y}) = \emptyset. \quad (4)$$

In other words, a single-deletion-error correcting code is a single-insertion-error correcting code.

Proof: Suppose two binary strings \mathbf{x} and \mathbf{y} of length n satisfy (3) but not (4). Suppose inserting x' in position i of \mathbf{x} and y' in position j of \mathbf{y} results in the same sequence of length $n + 1$. Without loss of generality, we assume $i \leq j$. This implies that deleting x_{j-1} and y_i from \mathbf{x} and \mathbf{y} respectively, would result in the same sequence of length $n - 1$. This contradicts (3), thus proving the lemma. ■

Step 3: Our extended codeword can detect up to two shift errors. In this step, we show that using the six delimiter bits and the checksum condition of (2), we can detect up to two shift errors. Consider m consecutive magnetic domains in racetrack memory. If no error happens, we need $m - 1$ shifts to access and read the m bits stored in these m domains. We summarize the resulting observations in positions $m - 5, m - 4, \dots, m$ for all possible error combinations in Table I. In Appendix B, we explain how Table I is obtained.

The most interesting case is when a combination of a deletion error and a repetition error takes place prior to the delimiter bits. In this case, we observe the delimiter bits as

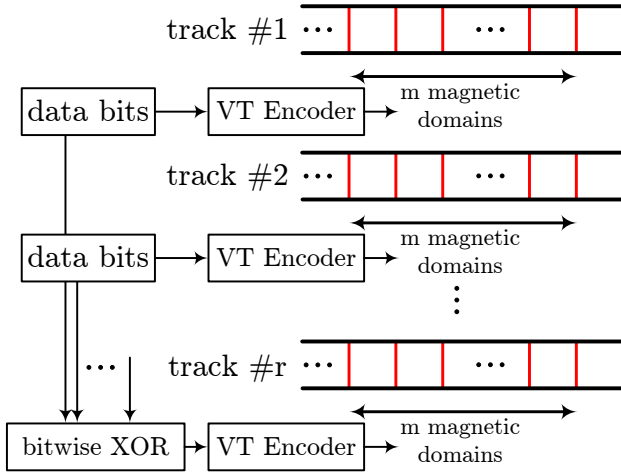


Fig. 2. Using an outer code on top of the code presented in Section IV allows us to correct two shift errors.

the error-free case, *i.e.* 110000. To detect such errors, we have the following result for VT codes.

Lemma 3. *For any two binary strings $\mathbf{x}, \mathbf{y} \in VT(n)$, \mathbf{y} cannot be obtained from \mathbf{x} by a deletion followed by an insertion (or by an insertion followed by a deletion).*

Proof: Suppose $\mathbf{x}' \in D_{-1}(\mathbf{x})$ and $\mathbf{y}' \in D_{-1}(\mathbf{y})$. If \mathbf{y} can be obtained from \mathbf{x} by a deletion followed by an insertion, then

$$D_{+1}(\mathbf{x}') \cap D_{+1}(\mathbf{y}') \neq \emptyset. \quad (5)$$

In other words, \mathbf{x}' and \mathbf{x}' do not satisfy (4) which in turn implies that \mathbf{x} and \mathbf{y} do not satisfy (3). However, this contradicts Lemma 1. This contradiction proves the result. A similar argument can be used when an insertion is followed by a deletion. ■

Lemma 3 immediately results in the following observation.

Corollary 1. *A combination of a deletion error and an insertion error on $\mathbf{x} \in VT(n)$ either results in \mathbf{x} or in $\mathbf{y} \notin VT(n)$ (an invalid codeword).*

The shaded row in Table I corresponds to a combination of a deletion error and an insertion error that resulted in an invalid codeword.

Step 4: Our extended codeword can correct a single shift error and decode the data. First, we need to understand how to recover data in the error-free case.

Error-free decoding: If no error is detected using the delimiter bits and the checksum, the $k = n - \ell - 1$ data bits are simply retrieved from the positions in \mathbf{c} that are not powers of two, *i.e.* positions 3, 5, 6, 7, 9, etc. So the only remaining task is to correct a shift error when it happens.

Correcting a deletion error: Deletion errors can be corrected using the elegant algorithm proposed by Levenshtein [16], [17]. This algorithm is presented below, and we refer the reader to [18] for a mathematical analysis.

- 1) Suppose a codeword $\mathbf{c} = (c_1, c_2, \dots, c_n)$ from $VT(n)$ is stored in racetrack memory, and we detect a single deletion, and we observe $\mathbf{c}' = (c'_1, c'_2, \dots, c'_{n-1})$.
- 2) Set ω to be the Hamming weight (number of 1's) of \mathbf{c}' .
- 3) Calculate the checksum $\sum_{i=1}^{n-1} ic'_i$ and set s to be the minimum amount that needs to be added to the checksum in order to make it $0 \bmod n + 1$.
- 4) If $s \leq \omega$, we restore the codeword by adding a 0 immediately to the left of the rightmost s 1's. Otherwise, we restore the codeword by adding a 1 immediately to the right of the leftmost $(s - \omega - 1)$ 0's.

Using this algorithm, we reconstruct the correct codeword and recover the dataword as discussed for the error free case. Note that while the decoding algorithm corrects the codeword, it does not reveal where the deletion occurred.

Correcting an insertion error: From Lemma 2, we know that VT codes can correct an insertion. The corresponding Insertion Correction Algorithm is a modified version of the Deletion Correction Algorithm presented above and is omitted here due to space limitations.

Step 5: The outer code corrects two shift errors. Suppose we detect two shift errors and the data is erased as in Table I. We can use an outer code to recover data in this case as illustrated in Fig. 2 where the outer code is a simple single parity code. We note that per track we can correct single shift errors and detect up to two shift errors.

APPENDIX A PROOF OF LEMMA 1

In this appendix, we prove Lemma 1. To prove the result, it suffices to show that after a single deletion, Varshamov-Tenengolts codes cannot be confused and unique recovery is guaranteed. We borrow from the argument given in [18].

Suppose a codeword $\mathbf{c} = (c_1, c_2, \dots, c_n)$ from $VT(n)$ is stored in Racetrack memory; and bit $x \in \{0, 1\}$ in position $1 \leq p \leq n$ is deleted to observe $\mathbf{c}' = (c'_1, c'_2, \dots, c'_{n-1})$. Let there be L_0 0's and L_1 1's to the left of x , and R_0 0's and R_1 1's to the right of x (with $p = 1 + L_0 + L_1$). Also, let ω denote the weight (number of 1's) of \mathbf{c}' . Calculate the new checksum $\sum_{i=1}^{n-1} ic'_i$.

Now if $x = 0$, the new checksum is R_1 less than $\sum_{i=1}^n ic_i$. We note that $R_1 \leq \omega$. If $x = 1$, the new checksum is $p + R_1$ less than $\sum_{i=1}^n ic_i$. We note that $p + R_1 = 1 + L_0 + L_1 + R_1 = 1 + \omega + L_0$ which is greater than ω .

So if the deficiency in the checksum is less than or equal to ω , we restore the codeword by adding a 0 just to the left of the rightmost R_1 1's. Otherwise, we restore the codeword by adding a 1 just to the right of the leftmost L_0 0's.

APPENDIX B DERIVATION OF TABLE I

In this appendix we describe how the combination of a VT code and six delimiter bits can be used to identify

two deletions, or a deletion and a repetition. The other cases appearing in Table I can be derived using similar arguments.

Two deletions: There are three cases to consider as listed below.

- 1) **Two deletions in the VT code:** In this case, in positions $m - 5, m - 4, m - 3$, and $m - 2$ we observe 0000 and position $m - 1$ corresponds to the first bit of the next extended codeword. In this case, we declare erasure and we use the outer-code to resolve this case.
- 2) **Two deletions in the delimiter bits:** For this scenario, the VT code is unaffected. However, we will declare either one or two deletions as we cannot identify the location of these errors. In positions $m - 5, m - 4, m - 3$, and $m - 2$ we can observe three possibilities as described below and position $m - 1$ corresponds to the first bit of the next extended codeword.
 - a) 0000: Similar to the case when two deletions happen in the VT code, we declare an erasure. The VT code is not affected but since we cannot determine that the errors were within the delimiter bits, we cannot trust the checksum. We declare erasure and we use the outer-code to resolve this case.
 - b) 1000: According to Table I, we declare a single deletion. Again note that the VT code is not affected here. So even if we assume a single deletion, data can be reliably recovered. The second deletion error will be detected when we reach the next set of delimiter bits are obtained. We note that our assumption of at most two shift errors every $m + 3$ shifts is needed to guarantee that the second shift error will be detected.
 - c) 1100: According to Table I, this is an error free case. Since the VT code is not affected here, data can be reliably recovered. However, if there was no error at all, positions $m - 1$ and m would correspond to the last two of the six delimiter bits. On the other hand, here positions $m - 1$ and m correspond to the first two bits of the next extended codeword and we deal with that when the next set of delimiter bits are obtained.
- 3) **One deletion in the VT code and one deletion in the delimiter bits:** For this case, in positions $m - 5, m - 4, m - 3$, and $m - 2$ we can observe two possibilities as described below and position $m - 1$ corresponds to the first bit of the next extended codeword.
 - a) 0000: Similar to the case when two deletions happen in the VT code, we declare an erasure. The VT code only contains a single deletion but since we cannot determine this fact, we cannot trust the checksum. We use the outer-code to resolve this case.
 - b) 1000: According to Table I, we declare a single deletion. Since the VT code only contains a single shift error, data can be reliably recovered.

One deletion and one repetition: We discussed the case when one deletion and one repetition happen within the VT code in Step 3 of Section IV. Suppose one deletion happens in the VT code and one repetition in the delimiter bits. For this case, in

positions $m - 5, m - 4, m - 3$, and $m - 2$ we can observe two possibilities as described below. The case in which one repetition happens in the VT code and one deletion in the delimiter bits can be explained similarly.

- 1) 1000: According to Table I, we declare a single deletion. Since the VT code only contains a single shift error, data can be reliably recovered.
- 2) 1100: If the checksum is zero modulo $n + 1$, we were lucky and the deletion followed by introducing the first bit of the delimiter bits in position $m - 6$ resulted in the correct codeword. If the checksum is not zero modulo $n + 1$, we declare erasure and we use the outer-code to resolve this case.

REFERENCES

- [1] S. Mittal, "A survey of techniques for architecting processor components using domain wall memory," *ACM Journal on Emerging Technologies in Computing Systems*.
- [2] L. Thomas, S.-H. Yang, K.-S. Ryu, B. Hughes, C. Rettner, D.-S. Wang, C.-H. Tsai, K.-H. Shen, and S. S. Parkin, "Racetrack memory: a high-performance, low-cost, non-volatile memory based on magnetic domain walls," in *IEEE International Electron Devices Meeting (IEDM)*, pp. 24–2, 2011.
- [3] Z. Sun, W. Wu, and H. Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2013.
- [4] R. G. Gallager, "Sequential decoding for binary channels with noise and synchronization errors," tech. rep., DTIC Document, 1961.
- [5] R. L. Dobrushin, "Shannon's theorems for channels with synchronization errors," *Problemy Peredachi Informatsii*, vol. 3, no. 4, pp. 18–36, 1967.
- [6] J. Ullman, "On the capabilities of codes to correct synchronization errors," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 95–105, 1967.
- [7] S. N. Diggavi and M. Grossglauser, "On transmission over deletion channels," in *the Annual Allerton Conference on Communication Control and Computing*, vol. 39, pp. 573–582, 2001.
- [8] A. Kavcic and R. Motwani, "Insertion/deletion channels: Reduced-state lower bounds on channel capacities," in *IEEE International Symposium on Information Theory*, pp. 229–229, 2004.
- [9] S. Diggavi, M. Mitzenmacher, and H. Pfister, "Capacity upper bounds for deletion channels," in *Proceedings of the International Symposium on Information Theory (ISIT)*, pp. 1716–1720, 2007.
- [10] M. Mitzenmacher, "A survey of results for deletion channels and related synchronization channels," *Probability Surveys*, vol. 6, pp. 1–33, 2009.
- [11] K. A. Abdel-Ghaffar, H. C. Ferreira, and L. Cheng, "Correcting deletions using linear and cyclic codes," *IEEE Transactions on Information Theory*, vol. 56, no. 10, pp. 5223–5234, 2010.
- [12] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu, "Hi-Fi playback: Tolerating position errors in shift operations of racetrack memory," in *Proceedings of ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 694–706, 2015.
- [13] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Avtomatika i Telemekhanika (in Russian)*, vol. 26, no. 2, pp. 288–292, 1965.
- [14] Z. Liu and M. Mitzenmacher, "Codes for deletion and insertion channels with segmented errors," *IEEE Transactions on Information Theory*, vol. 56, no. 1, pp. 224–232, 2010.
- [15] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Transactions on Computers*, vol. 43, no. 1, pp. 68–77, 1994.
- [16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet physics doklady*, vol. 10, p. 707, 1966.
- [17] V. Levenshtein, "Binary codes capable of correcting spurious insertions and deletions of ones," *Problems of Information Transmission*, vol. 1, no. 1, pp. 8–17, 1965.
- [18] N. J. Sloane, "On single-deletion-correcting codes," *Codes and Designs*, pp. 273–291, 2002.